

# Improving the integration of task nesting and dependencies in OpenMP

Josep M. Perez\*, Vicenç Beltran\*, Jesus Labarta\*<sup>†</sup> and Eduard Ayguadé\*<sup>†</sup>

{josep.m.perez, vbeltran, jesus.labarta, eduard.ayguade}@bsc.es

ORCID: 0000-0002-0558-7600, 0000-0002-3580-9630, 0000-0002-7489-4727, 0000-0002-5146-103X

\*Barcelona Supercomputing Center (BSC), C/ Jordi Girona 29, 08034-Barcelona, Spain

<sup>†</sup>Universitat Politècnica de Catalunya, Campus Nord, Ed. D6, C/ Jordi Girona 1-3, 08034-Barcelona, Spain

**Abstract**—The tasking model of OpenMP 4.0 supports both nesting and the definition of dependences between sibling tasks. A natural way to parallelize many codes with tasks is to first taskify the high-level functions and then to further refine these tasks with additional subtasks. However, this top-down approach has some drawbacks since combining nesting with dependencies usually requires additional measures to enforce the correct coordination of dependencies across nesting levels. For instance, most non-leaf tasks need to include a *taskwait* at the end of their code. While these measures enforce the correct order of execution, as a side effect, they also limit the discovery of parallelism.

In this paper we extend the OpenMP tasking model to improve the integration of nesting and dependencies. Our proposal builds on both formulas, nesting and dependencies, and benefits from their individual strengths. On one hand, it encourages a top-down approach to parallelizing codes that also enables the parallel instantiation of tasks. On the other hand, it allows the runtime to control dependencies at a fine grain that until now was only possible using a single domain of dependencies.

Our proposal is realized through additions to the OpenMP task directive that ensure backward compatibility with current codes. We have implemented a new runtime with these extensions and used it to evaluate the impact on several benchmarks. Our initial findings show that our extensions improve performance in three areas. First, they expose more parallelism. Second, they uncover dependencies across nesting levels, which allows the runtime to make better scheduling decisions. And third, they allow the parallel instantiation of tasks with dependencies between them.

**Index Terms**—computer languages; runtime library; OpenMP; task nesting; task dependencies; weak dependencies; weakwait; taskwait; single dependency domain; top-down programming; task decomposition;

## I. INTRODUCTION

One of the challenges of parallel programming is the coordination of the joint activity of several cooperating tasks. OpenMP [1] and Cilk++[2] support the fork-join model to exploiting structured parallelism. In the fork-join model, several independent tasks are spawned and executed until they reach an implicit synchronization point at the end of the fork-join construct.

However, several studies have identified limitations on the OpenMP fork-join execution model [3], especially for exploiting irregular parallelism. The OpenMP tasking model [4], [5] was developed to exploit irregular and nested parallelism in the presence of complicated control structures or recursion. In OpenMP, independent tasks are created and executed in parallel, until a *taskwait* – an explicit synchronization point – is reached.

The tasking model is more flexible and suitable than the fork-join model to exploit irregular and nested parallelism. However, it lacks a mechanism for fine-grained synchronization. To overcome this issue, a data-flow model was proposed by Duran et al.[6] and introduced in OpenMP 4.0. To support it in the language, the *depend* clause was added to the *task* construct. From the point of view of its semantics, each task defines an inner, unique and independent domain on which to calculate the dependencies of its direct children.

However, since the dependency domains are disconnected, codes that combine nesting and dependencies need to perform additional actions to coordinate dependencies across nesting levels. These measures, as a side effect, reduce the amount of exploitable parallelism and delay its discovery.

Our work is a natural extension of the previous research that addresses the problem of combining task nesting with fine-grained dependencies between tasks. It enhances the data-flow model of OpenMP by supporting fine-grained dependencies not only between sibling task but between tasks with any family relationship. By doing this, it eliminates the drawbacks caused by the mechanisms used to coordinate dependencies across nesting levels.

This document is organized as follows. We begin in Section II by describing briefly how task dependencies work in OpenMP. Section III describes the interaction between dependencies and task nesting and their current limitations, which is the motivation for this work. In Section IV we present a new type of *taskwait* with new semantics.

The integration of nesting and dependencies consists of two halves. The first half, described in Section V, extends the semantics of the new *taskwait* to allow inner tasks to release dependencies that cross the domain of their parent task. Section VI completes the integration by allowing dependencies to propagate into the inner dependency domain of tasks and thus to reach subtasks. The powerfulness of the proposals and the scope of applicability can be improved by combining them with techniques of Sections V and VI. This is discussed in Section VII.

Next we evaluate our proposals with several benchmarks in Section VIII. We show the codes and the impact that our proposals have on them. Finally, we present our conclusions in Section IX and future directions in Section X.

## II. OPENMP TASKS WITH DEPENDENCIES

OpenMP introduced tasks in version 3.0 and added support for task dependencies in version 4.0. In this section we briefly describe the OpenMP tasking model and define some terms that we use throughout the rest of this text to reason about it.

OpenMP supports tasks through the *task* construct. In the C language it consists of the following pragma followed by a statement:

```
#pragma omp task optional clauses
```

The statement can be either a single statement, or a C compound statement, which is a set of statements enclosed in braces. In addition, statements can be either regular C statements, or OpenMP directives and constructs, including the task construct itself.

In the general case, the statement that follows the pragma is to be executed asynchronously. Whenever a thread encounters a task, it instantiates it and resumes its execution after the construct. The task instance can be executed either by that same thread at some other time or by another thread.

The semantics can be altered through additional clauses and through the properties of the enclosing environment in which the task is instantiated. Those details and which threads can execute a given task are described in the OpenMP specification [7].

Dependencies allow tasks to be scheduled after other tasks. OpenMP uses the *depend* clause to define them. The contents consist of either the *in*, *out* or *inout* keyword followed by a colon and a comma separated list of elements.

Tasks that contain an *out* or *inout* element are delayed until all sequentially preceding tasks with the same element in a *depend* clause have finished. The *in* elements defer the task until all sequentially preceding tasks with the same element as *out* or *inout* have finished.

The scope of the dependency calculation is restricted to that determined by the enclosing (possibly implicit) task. That is, the contents of the *depend* clause of two tasks can determine dependencies between them only if they share the same parent task. In this sense, tasks define an inner and independent *dependency domain* into which to calculate the dependencies between its direct children.

## III. COMBINING TASK NESTING AND DEPENDENCIES

The OpenMP 4.5 standard supports task nesting and task dependencies. Being able to combine both features is important for programmability. Listing 1 shows an example that combines nesting and dependencies with two levels of tasks. For brevity and without loss of generality, the inner tasks do not have dependencies between their siblings.

To parallelize the original code using a top-down approach we would perform the following steps. First, we would add the pragmas of the outer tasks. Since they have conflicting accesses over some variables, we add the *depend* clause. In general it is a good practice to have entries to protect all the accesses of a task. Doing this reduces the burden on the programmer, improves the maintainability of the code and reduces the chances to overlook conflicting accesses.

```
#pragma omp task depend(inout: a, b) // Task T1
{
    a++; b++;
    #pragma omp task depend(inout: a) // Task T1.1
    a += ...;
    #pragma omp task depend(inout: b) // Task T1.2
    b += ...;
    #pragma omp taskwait
}

#pragma omp task depend(in: a, b) depend(out: z, c, d) // Task T2
{
    z = ...;
    #pragma omp task depend(in: a) depend(out: c) // Task T2.1
    c = ... + a + ...;
    #pragma omp task depend(in: b) depend(out: d) // Task T2.2
    d = ... + b + ...;
    #pragma omp taskwait
}

#pragma omp task depend(in:a, b, d) depend(out:e, f) // Task T3
{
    #pragma omp task depend(in:a, d) depend(out:e) // Task T3.1
    e = ... + a + d + ...;
    #pragma omp task depend(in:b) depend(out:f) // Task T3.2
    f = ... + b + ...;
    #pragma omp taskwait
}

#pragma omp task depend(in: c, d, e, f) // Task T4
{
    #pragma omp task depend(in: c, e) // Task T4.1
    ... = ... + c + e + ...;
    #pragma omp task depend(in: d, f) // Task T4.2
    ... = ... + d + f + ...;
    #pragma omp taskwait
}
```

Listing 1. A code with nesting and dependencies.

Then inside each task, we identify separate functionalities, convert each into a task, and add a *taskwait* at the end of each outer task. If we follow the same approach, the *depend* clause of each inner task will contain entries to protect its own accesses.

When we consider how the *depend* clauses are composed, we observe that outer tasks contain a combination of elements to protect their own accesses and elements that are only needed by their subtasks. This is necessary to avoid data-races between subtasks with different parents. In this sense, the latter defer the execution of the outer task until all the dependencies of its subtasks have been fulfilled. In addition, the *taskwait* at the end of each outer task delays the release of its dependencies until its subtasks have finished.

The inclusion in the *depend* clause of elements only required by subtasks and the *taskwait* at the end effectively link the dependency domain of the task with that of its subtasks, which otherwise would be disconnected. However, the elements of the *depend* clause that the task does not need for itself delay the execution of the task and thus the instantiation of its subtasks. Moreover, the *taskwait* causes the whole set of dependencies to be released at once. Hence, these two aspects hinder the discovery of parallelism, by delaying and partially hiding it.

The combination of nesting and dependencies has three aspects that can be improved:

- 1) The presence of the `taskwait` directive delays the completion of the enclosing task and thus the release of the system or user-level thread and its stack.
- 2) A task with subtasks cannot release incrementally its own dependencies and those of its subtasks. Instead they are all released together once the task and all of its subtasks have finished.
- 3) The presence of elements in the `depend` clause that are only needed by subtasks defers their instantiation even when only their execution needs to be deferred.

Some of these limitations could be avoided by not using nesting. For instance, the outer level of tasks of listing 1 could be eliminated by removing the pragmas. Notice that while in the example code this simple change is enough, other codes may require more complex transformations.

Figure 1a shows the graph of listing 1 with two levels of tasks, and 1b the graph after removing the outer level and the `taskwait`s. Each task is represented as a rectangle that contains the name in the comment of the listing, and its subtasks if any. The edges indicate dependencies and are labeled by the variable involved on the `depend` clause.

Each version of the code has advantages over the other. On one hand, the original code with two levels of tasks:

- Is more natural when programming in a top-down manner.
- Can create tasks in parallel. For instance, the subtasks of T2 can be created in parallel to the subtasks of T3.
- Given a limited lookahead window, it can detect distant coarse-grained parallelism.
- Has the opportunity of reducing the overhead by not deferring the execution of inner tasks.

On the other hand, the flat code has the following advantages:

- The instantiation of the inner tasks is not delayed by dependencies on outer tasks.
- Its tasks only wait for their exact dependencies. For instance task T1.2 does not defer neither T2.1 nor T3.1.
- It does not suffer the extra overhead of the additional `taskwait`s.

In this paper we improve the integration of nesting with dependencies in a way that preserves the programmability and advantages of each approach. Our approach preserves the parallel generation of work from nesting and the ability to detect distant parallelism. From dependencies it retains the fine-grained control of dependencies that was only possible by programming with a single-level of tasks with dependencies.

#### IV. DETACHING THE TASKWAIT AT THE END OF THE TASK FROM THE TASK CODE

Codes with nested tasks usually have a `taskwait` at the end of each non-leaf task. As shown in the previous section, this is necessary when combining nesting with dependencies.

Some programming models include an equivalent implicit synchronization at the end of their tasks. For instance Cilk Plus[2] includes an implicit `cilk_sync` at the end of functions

that spawn tasks. In Cilk, this `sync` operation is performed from within the function code, after the destruction of the local C++ objects, but before returning. In OpenMP there is no implicit behavior equivalent to a `taskwait`. Instead, OpenMP defines the `taskgroup` construct. The construct does not allow the execution to continue until all deeply nested tasks within its scope have finished.

In this section we propose to replace some uses of the `taskwait` at the end of tasks with a new mechanism with different semantics. To this end, we extend the `task` construct with the `wait` clause. Tasks with this clause will perform a `taskwait`-like operation immediately *after* exiting from the task code. Since it is performed outside the scope of the code of the task, this happens once the task has abandoned the stack. For this reason, its use is restricted to tasks that upon exiting do not have any subtask accessing its local variables. Otherwise, the regular `taskwait` shall be used instead.

The new semantics allow tasks to release their stack. Moreover, once subtasks have finished, the runtime does not need to switch back to the task code. Whereas the equivalent code with a `taskwait` would require to resume the task code, just to return from it and thus to release the stack.

In addition, the new semantics make the runtime aware of the fact that the task code has ended and thus will not create more subtasks. The runtime can take advantage of this information to perform optimizations. The proposal of the following section is built on top of this knowledge.

#### V. FINE-GRAINED RELEASE OF DEPENDENCIES ACROSS NESTING LEVELS

Detaching the `taskwait` at the end of a task from the task code allows the runtime to be made aware earlier of the fact that the task code has finished and that it will not create further subtasks. In an scenario with task nesting and dependencies, this knowledge allows it to make assumptions about dependencies. Since the task code is finished, the task will no longer perform by itself any action that may require the enforcement of its dependencies. Only the dependencies needed by its live subtasks need to be preserved. In most cases, these dependencies are the ones associated to an element of the `depend` clause that also appears in a live subtask.

Therefore, the dependencies that do not need to be enforced anymore could be released. For instance, the code in listing 2 contains a task T1 with two subtasks and then two other tasks that depend on the first and its subtasks. When T1 exits from its code, if T1.1 has still not finished, the dependency from T1 to T2 becomes a dependency from T1.1 to T2. That is, T2 will become logically ready as soon as T1.1 finishes, since that is the only live subtask of T1 with a `depend` clause that defers T2.

In this sense, when a task with the `wait` clause exits from its code, the effects over the dependencies are equivalent to replacing the effects of its `depend` clause by that of the sequence of its unfinished subtasks. Moreover, this is equivalent to merging its inner dependency domain into that of its parent.

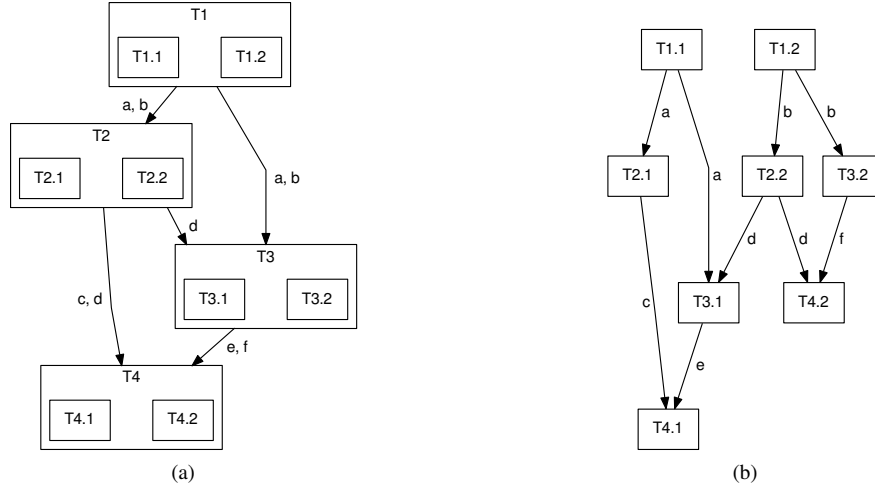


Fig. 1. Graph of the example code: (a) original form and (b) after removing the outer level of tasks.

```

#pragma omp task depend(inout:a, b) wait           // Task T1
{
    a++; b++;
    #pragma omp task depend(inout:a)               // Task T1.1
    a += ...;
    #pragma omp task depend(inout:b)               // Task T1.2
    b += ...;
}

#pragma omp task depend(in:a)                       // Task T2
... = ... + a + ...;
#pragma omp task depend(in:b)                       // Task T3
... = ... + b + ...;

```

Listing 2. A code with task nesting, dependencies and the wait clause.

To enable tasks to release their dependencies in this way we propose to extend the *task* construct with the *weakwait* clause. The clause is an alternative to the *wait* clause that indicates that the dependencies can be released incrementally as described above.

Notice that this improvement is only possible once the runtime is aware of the end of the code of a task. However, it may also be desirable to trigger this mechanism earlier. For instance, a task may use certain data only at the beginning and then perform other lengthy operations that delay the release of the dependencies associated to that data.

To cover this need we propose a new directive to assert that a task will no longer perform accesses that conflict with part of the contents of its *depend* clause. The new directive has the following form:

```
#pragma omp release depend(...)
```

The contents of the *depend* clause is a subset of that of the *task* construct that is no longer referenced in the rest of the lifetime of the task and its future subtasks.

## VI. WEAK DEPENDENCIES

The previous sections allow subtasks to expose their release of dependencies to the outer task levels. In other words, they

propagate dependencies outwards. However, these dependencies are fine-grained, and the tasks that are deferred by those, since they are usually of an outer level, are likely to have coarser dependencies. Thus, in many cases, these mechanisms alone will not be enough to make more dependency-controlled parallelism available.

For instance, the code of listing 1 after converting its tasks to use the *weakwait* clause, does not allow T2.1 to become ready as soon as T1.1 has finished. The reason is that T2 depends on both T1.1 and T1.2. Therefore it will not start until both have finished, and thus T2.1 will not exist until T2 has had a chance to create it. This is a consequence of having in the *depend* clause of T2 elements that only are needed for its subtasks.

Some of the elements of the *depend* clause may be needed by the task itself, others may be needed only by its subtasks, and others may be needed by both. The ones that are only needed for the subtasks only serve as a mechanism to link the outer domain of dependencies to the inner one. In this sense, allowing them to defer the execution of the task is unnecessary, since the task does not actually perform any conflicting accesses by itself.

For this reason we propose to extend the *depend* clause with three additional dependency types: *weakin*, *weakout* and *weakinout*. Their semantics are analogous to the ones without the *weak* prefix. However, the weak variants indicate that the task does not perform by itself any action that requires the enforcement of the dependency. Instead those actions can be performed by any of its deeply nested subtasks. Any subtask that may directly perform those actions needs to include the element in its *depend* clause in the non-weak variant. In turn, if the subtask delegates the action to a subtask, the element must appear in its *depend* clause using at least the weak variant.

Weak variants do not imply a direct dependency, and thus do not defer the execution of tasks. Their purpose is to serve as linking point between the dependency domains of each nesting level. Until now, out of all the tasks with the same parent, the first one with a given element in its *depends* clause

was assumed to not have any input dependency. However, this assumption is no longer true since the dependency domains are no longer isolated. Instead, if the parent has that same element with a *weak* dependency type, there may actually be a previous and unfinished task with a *depend* clause that has a dependency to it. If we calculated dependencies as if all types were non-weak, in such a case, the source of the dependency, if any, would be the source of the non-enforced dependency on its parent over the same element.

This change, combined with the fine-grained release of dependencies, merges the inner dependency domain of a task into that of its parent. Since this happens at every nesting level, the result is equivalent to an execution in which all tasks had been created in a single dependency domain.

To illustrate this, listing 3 shows the initial code after applying all the techniques and figure 2 its graph at various stages. The outer tasks produce the graph shown in figure 2a. The dashed edges indicate the dependencies that would be in place if the corresponding elements of the *depend* clause had not been weak. Since there is no actual dependency in that graph, its tasks can run in parallel, and therefore they can instantiate their subtasks in parallel too.

Figure 2b shows the graph after instantiating the subtasks but before returning from the outer tasks. Notice that the input dependencies of the inner tasks originate from the outer tasks since the corresponding elements of the *depend* clause of their parents have weak dependency types.

Once the outer tasks return from their code, the fine-grained release of dependencies is triggered, which transforms the graph into the one shown in figure 2c. The outer tasks have been kept grayed out for reference. Notice that this graph is equivalent to the one obtained initially after removing the outer layer of tasks. That is, the behavior is equivalent as if the tasks had been instantiated in the same domain of dependencies.

## VII. EXTENDING THE APPLICABILITY

The example that illustrates the previous sections is a simple code with very few variables. Task nesting occurs naturally when solving a problem with a task that divides the problem into subproblems, and each is handled by a subtask. Usually, subdividing a problem implies a division of the data. However, OpenMP specifies that array sections that appear in the *depend* clause must be either fully overlapping or non-overlapping at all. In other words, it does not allow partially overlapping array sections.

If the data that determines the dependencies is the one that is divided, this restriction limits the problem decompositions to a fixed number of subtasks. For instance, the *axpy* operation consists in calculating  $y \leftarrow \alpha x + y$  for two vectors  $x$  and  $y$ . Listing 4 shows a task implementation over double precision arrays of  $N$  elements. Notice that this code subdivides the operation into 4 tasks. The length of the array section that each subtask covers depends on the value of  $N$ , but the number of subtasks cannot be changed, since each subtask section must also appear in the *depend* clause of the *axpy* task. Therefore, dividing the problem into more subtasks would either require

```
#pragma omp task depend(inout: a, b) weakwait // Task T1
{
    a++; b++;
    #pragma omp task depend(inout: a) // Task T1.1
    a += ...;
    #pragma omp task depend(inout: b) // Task T1.2
    b += ...;
}

#pragma omp task depend(out: z) depend(weakin: a, b) \
depend(weakout: c, d) weakwait // Task T2
{
    z = ...;
    #pragma omp task depend(in: a) depend(out: c) // Task T2.1
    c = ... + a + ...;
    #pragma omp task depend(in: b) depend(out: d) // Task T2.2
    d = ... + b + ...;
}

#pragma omp task depend(weakin: a, b, d) depend(weakout: e, f)
weakwait // Task T3
{
    #pragma omp task depend(in: a, d) depend(out: e) // Task T3.1
    e = ... + a + d + ...;
    #pragma omp task depend(in: b) depend(out: f) // Task T3.2
    f = ... + b + ...;
}

#pragma omp task depend(weakin: c, d, e, f) weakwait // Task T4
{
    #pragma omp task depend(in: c, e) // Task T4.1
    ... = ... + c + e + ...;
    #pragma omp task depend(in: d, f) // Task T4.2
    ... = ... + d + f + ...;
}
```

Listing 3. A code with nesting, dependencies, fine-grained release of dependencies and weak accesses.

```
void axpy(double *x, double *y, double alpha, int N) { // Task size
    int S = (N + 4 - 1) / 4;
    int lastS = N - 3*S;

    #pragma omp task weakwait \
    depend(weakin: x[0:S], x[S:S], x[2*S:S], x[3*S:lastS]) \
    depend(weakinout: y[0:S], y[S:S], y[2*S:S], y[3*S:lastS])
    for (int step = 0; step < 4; step++) {
        int start = step*S;
        int end = (step < 3 ? (step+1)*S : N);
        int count = end - start;

        #pragma omp task \
        depend(in: x[start:count]) depend(inout: y[start:count])
        for (; start < end; start++)
            y[start] += alpha * x[start];
    }
}
```

Listing 4. Implementation of the *axpy* operation as a task with weak accesses that decomposes the operation into 4 subtasks.

a specific implementation for that number of subtasks or abandoning the use of weak dependencies and the fine-grained release of dependencies.

Calculating dependencies over partially overlapping arrays has already been proposed in [8]. When we remove the restriction, the previous code can be rewritten as shown in listing 5. The  $S$  variable determines the number of elements to

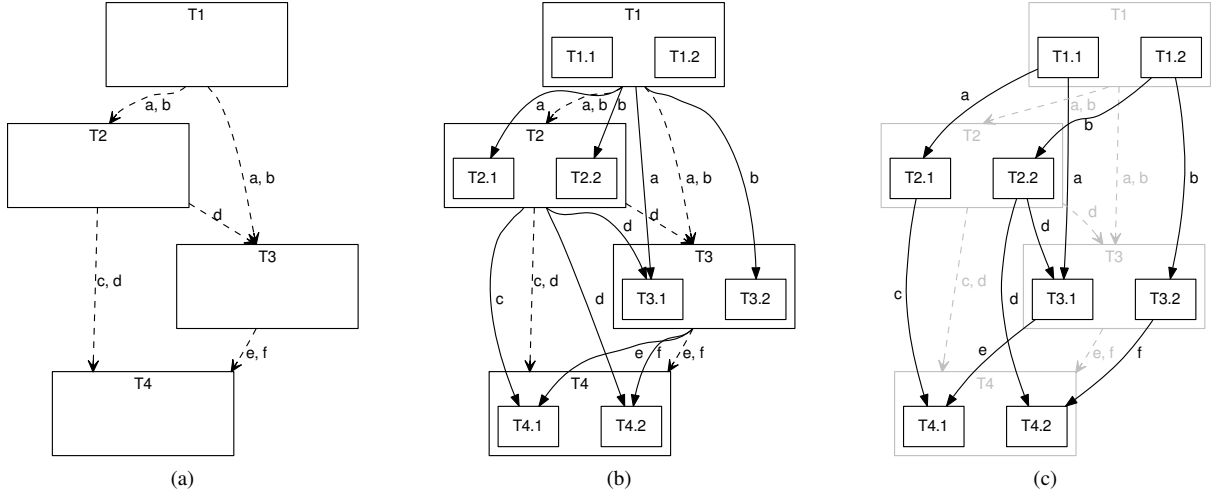


Fig. 2. Graph of the final code at 3 possible stages: (a) with only the outer level instantiated, (b) right before exiting the code of the outer tasks, and (c) right after exiting the code of the outer tasks.

```

void axpy(double *x, double *y, double alpha, int N) {
    // S is global and specifies the desired task size in elements

    #pragma omp task weakwait \
        depend(weakin: x[0:N]) depend(weakinout: y[0:N])
    for (int start = 0; start < N; start += S) {
        int count = min(S, N - start);

        #pragma omp task \
            depend(in: x[start:count]) depend(inout: y[start:count])
        for (int i = 0; i < count; i++)
            y[start + i] += alpha * x[start + i];
    }
}

```

Listing 5. Implementation of the axpy operation as a task with weak accesses that decomposes the operation into an arbitrary number of subtasks.

handle in each subtask. Notice that now the number of subtasks depends on the length of the arrays and  $S$ . In fact  $S$  could be a parameter of the function. In addition, the pragma of the outer task has been shortened and simplified.

## VIII. EVALUATION

To evaluate the impact of our proposals, we have implemented a new runtime called Nanos6 and used the Mercurium[9] source-to-source compiler with Nanos6 as the target runtime. The new runtime implements our proposals and supports dependencies over partially overlapping array sections. Our evaluation hardware is a Cavium ThunderX 2K blade[10]. The tests have been run on a single processor with up to 48 ARMv8 cores. We have used GCC 5.3.0 for both the runtime and the code emitted by Mercurium.

### A. Multiple AXPY

Our first benchmark performs 20 calls to the axpy function already shown in the previous section in listing 5. Every call is performed over the same pair of arrays. Therefore, there is a dependency between tasks of each call over the array on which the accumulation is performed.

TABLE I  
SUMMARY OF THE MULTIPLE AXPY SERIES.

Series	Nested	Dependencies		Synchronization between levels
		Outer	Inner	
<i>nest-weak-release</i>	yes	weak	regular	weakwait and release directive
<i>nest-weak</i>	yes	weak	regular	weakwait
<i>nest-depend</i>	yes	regular	regular	taskwait
<i>flat-depend</i>	no	—	regular	no
<i>flat-taskwait</i>	no	—	no	taskwait

We have written additional variants of this algorithm to evaluate the impact on performance of the contributions of this paper. The set of variants is the following:

- **nest-weak** The implementation already shown, that uses nesting, the *weakwait* clause, and weak dependencies in the outer tasks.
- **nest-weak-release** Identical to the previous one but in addition, the outer tasks use the *release* directive over the *inout* array after creating each subtask.
- **flat-depend** Implementation with dependencies but without the outer level of tasks.
- **flat-taskwait** Without the outer level of tasks, without dependencies, and isolating each call to axpy with a taskwait.
- **nest-depend** With dependencies, nesting and none of our proposals.

Table I summarizes the features of each variant.

We have measured the performance of each implementation with 48 cores and  $384 \times 2^{20}$  elements. Since task task granularity has an important effect on the availability of work and the overhead of the runtime, we have run the experiments with several task sizes. Figure 3 shows the total performance in the

top graph, and the second level data cache miss ratio in the bottom graph. Both graphs share the same horizontal axis. The lower axis indicates the number of array elements that each leaf task processes, and the upper axis its equivalent duration in a sequential execution.

The nest-depend and the flat-taskwait versions do not allow dependencies of the inner tasks to cross their outer level. Since the flat-taskwait version does not use dependencies, the performance difference between it and the nest-depend version is due to the overhead of calculating dependencies.

The flat-depend version has only the inner level of tasks, and thus makes the runtime aware of all the dependencies. The nest-weak versions also makes it aware through the use of the extensions of the previous sections. When a task finishes, the scheduler of the runtime can use this information to dispatch a successor to the same core. Since a dependency is likely to be associated to accesses to the same data, this policy is likely to improve temporal locality of the cache. For this reason, while the other versions are limited by the memory bandwidth, the versions that uncover the inner dependencies exploit the cache better. This effect is shown in the bottom graph of the figure, that shows lower level-2 data cache miss ratio.

The performance improvement of the nest-weak code over the flat-depend one is due to two factors. First, the instantiation of the inner tasks in parallel. And second, as a consequence the runtime is made aware of the dependencies between the inner tasks earlier. Hence it can start exploiting temporal locality earlier.

Finally, the nest-weak-release version uses the *release* directive to make this happen as soon as possible. Hence it is able to achieve better performance due to less cache misses.

Figure 4 shows the strong scalability with tasks of  $14 \times 2^{10}$  elements. The results indicate that the weak variants scale better than the rest.

### B. Gauss-Seidel

Our second benchmark is an application of the Gauss-Seidel algorithm to solve the propagation of heat over a plane. It is a bidimensional stencil algorithm that is applied over a rectangle iteratively. The operations of an iteration have dependencies to the operations of the same iteration and the previous one. Within an iteration, the dependencies are such that there is diagonal wavefront parallelism.

Listing 6 shows its main code. Each iteration of the algorithm is performed by a task, that then is divided in further subtasks. To enable the fine-grained release of dependencies, the outer task uses the *weakwait* clause. Since the outer task does not perform the actual accesses to the data, its *depend* clause uses weak dependency types. To parallelize the iteration, the data has been divided into square blocks of TS by TS elements, and each inner task performs the updates of a single block.

Like in the previous benchmark, we have made an additional implementation with only the inner level of tasks, and another with the two levels, dependencies, but none of the enhancements that we contributed. Figure 5 shows the performance of each

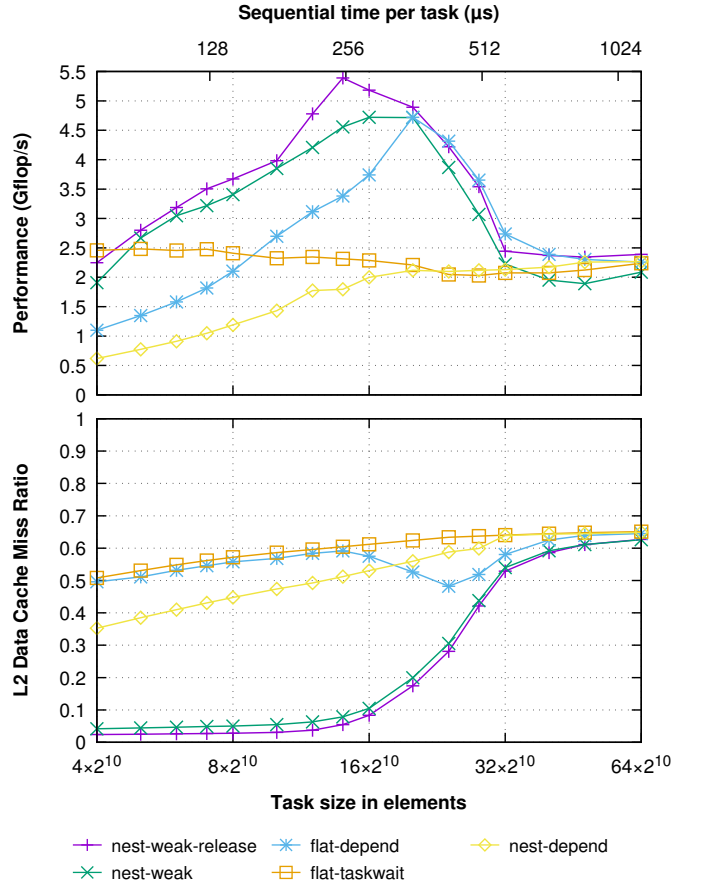


Fig. 3. Performance and level-2 data cache miss ratio with respect to the task size of 20 calls to the each implementation of axpy over the same vectors.

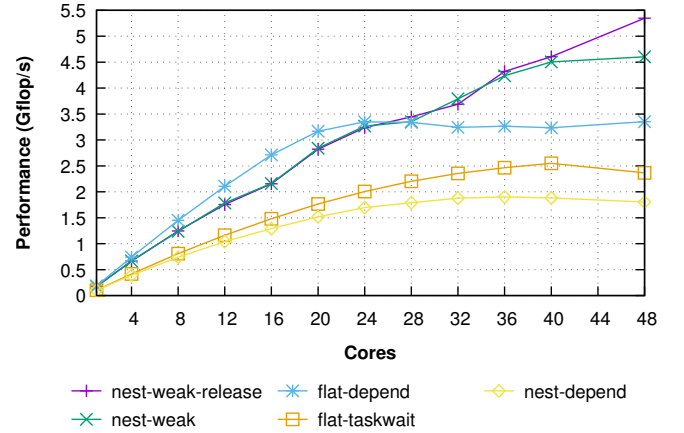


Fig. 4. Strong scalability of 20 calls to the each implementation of axpy over the same vectors with tasks of  $14 \times 2^{10}$  elements.

version with 48 cores, 48 iterations and several task sizes. The data is a square array of double precision floating point numbers of 27648 elements per side.

While there are dependencies between an iteration and the following one, there is also parallelism between the subtasks

```

double A[2+BLOCKS][2+BLOCKS][TS][TS];

for (int iteration = 0; iteration < IT; iteration++) {
    #pragma omp task weakwait depend(weakinout: A[:][:][:])
    for (long i=1; i <= BLOCKS; i++) {
        for (long j=1; j <= BLOCKS; j++) {
            #pragma omp task \
            depend(in: A[i-1][j][:]) /* Top */ \
            depend(in: A[i][j-1][:]) /* Left */ \
            depend(inout: A[i][j][:]) /* Center */ \
            depend(in: A[i][j+1][:]) /* Right */ \
            depend(in: A[i+1][j][:]) /* Bottom */ \
            tile_kernel(BLOCKS, TS, A, i, j);
        }
    }
}

```

Listing 6. Implementation of Gauss-Seidel with 2 levels of tasks, weak dependencies, weakwait, dependencies between sibling tasks of the same parent, and dependencies between tasks with different parents.

of different iterations. However, the nest-depend version is unable to exploit it due to the strict enforcement of the outer task dependencies. This is the main difference between its performance and that of the flat-depend version. The nest-weak version is able to find the fine-grained dependencies that cross the outer tasks and thus to extract as much parallelism as the flat-depend version, and can also generate the tasks of each iteration in parallel and thus perform better at smaller task granularities. In some cases it is able to achieve similar performance to the flat-depend code, but with tasks that are 4 times as small or even smaller.

In this benchmark, when we apply the *release* directive, it does not improve the performance. Instead it only adds overhead, even when we try to minimize it by releasing by horizontal panels instead of by blocks.

Figure 6 shows the strong scalability of each implementation with two tasks sizes. The top graph has tasks of  $64 \times 64$  elements, and the bottom graph tasks of  $128 \times 128$  elements. The implementations that do not use weak dependencies do scale beyond 8 cores with the smaller tasks, and 24 with the bigger tasks. When we apply weak dependencies, the code scales up to 48 cores.

### C. Quicksort followed by Prefix Sum

To illustrate the use on a recursive algorithm we have implemented a benchmark that performs a quicksort[11] over an array, and then a prefix sum[12]. Its main code is shown in listing 7. The array is called data, and it has N elements. The TS parameter determines the number of elements for the base case for the quicksort and the prefix sum.

The quicksort, which starts at line 8, cannot use weak dependencies since it needs to access the data to find the pivot and to perform the partition. Notice that unlike most implementations of the algorithm that use nesting, our version also uses dependencies. The purpose is to avoid a deep taskwait, and instead to allow the following algorithm to start as soon as parts of the data are moved to their final position.

The prefix sum operation has been implemented with recursion and weak dependencies. The data is divided in blocks of

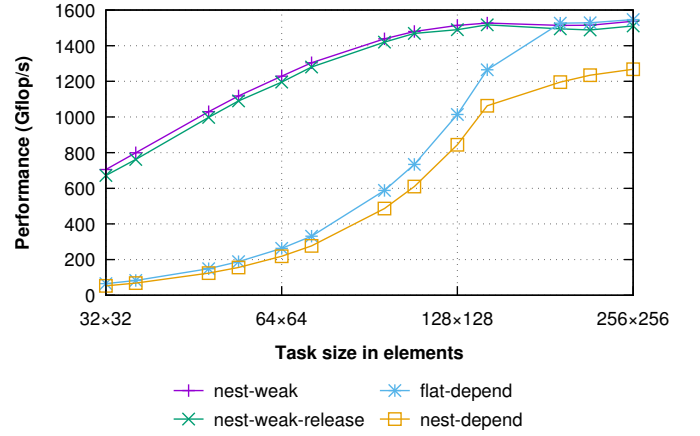


Fig. 5. Performance with respect to the task size of 48 iterations of Gauss-Seidel.

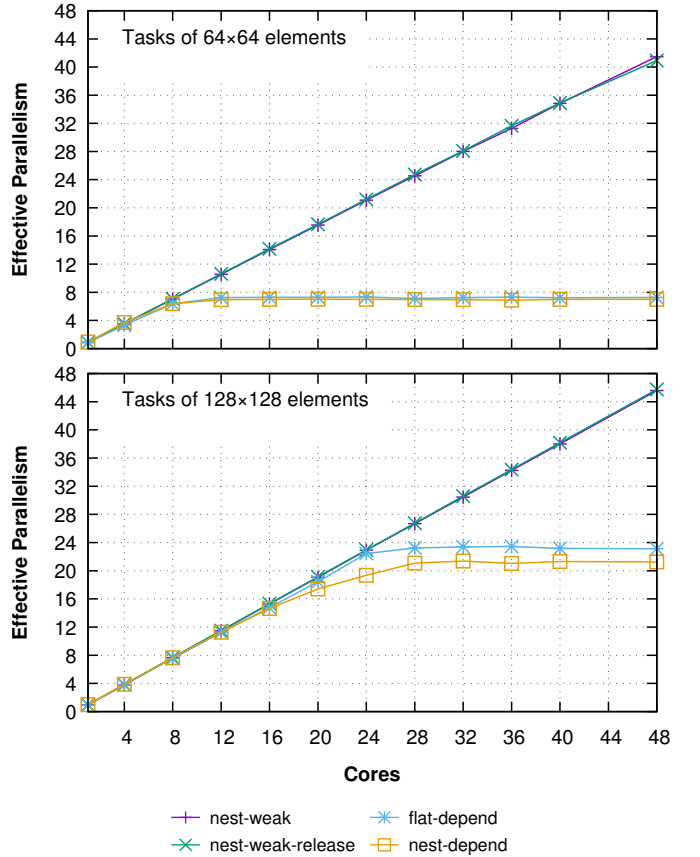


Fig. 6. Strong scalability of 48 iterations of Gauss-Seidel with tasks of  $64 \times 64$  (top), and  $128 \times 128$  (bottom) elements.

TS elements and then each block is solved by the base case, that is a non-recursive prefix sum.

Since each block will have in its last position the total contribution of the block, we perform a recursive call to the algorithm over those elements. Hence the algorithm is now applied with stride equal to the block size. This call calculates



```

1 #pragma omp task weakwait depend(inout: data[0:N])
2 quick_sort(data, N, TS);
3
4 #pragma omp task weakwait depend(weakinout: data[0:N])
5 prefix_sum(data, N, TS, 1);
6
7
8 void quick_sort(type *data, long N, long TS) {
9     // Base case
10    if (N <= TS) {
11        #pragma omp task depend(inout: data[0:N])
12        insertion_sort(data, N);
13        return;
14    }
15
16    type pivot = find_pivot(data, N);
17    long pivot_index = partition(data, pivot, N);
18
19    if (pivot_index > 1) {
20        #pragma omp task weakwait \
21        depend(inout: data[0:pivot_index])
22        quick_sort(data, pivot_index, TS);
23    }
24    if (pivot_index < N-1) {
25        #pragma omp task weakwait \
26        depend(inout: data[pivot_index:N-pivot_index])
27        quick_sort(data + pivot_index, N - pivot_index, TS);
28    }
29 }
30
31
32 void prefix_sum(type *data, long N, long TS, long stride) {
33     // Base case
34     if (N <= TS*stride) {
35         #pragma omp task depend(in: data[0]) \
36         depend(inout: data[stride:N-stride])
37         for (long i = stride; i < N; i += stride)
38             data[i] += data[i-stride];
39         return;
40     }
41
42     // Compute blocks independently (base case call)
43     for (long i=0; i < N; i += TS*stride) {
44         long size = min(TS*stride, N-i);
45         prefix_sum(data + i, size, TS, stride);
46     }
47
48     // Index of the last element of the first block
49     long substart = (TS - 1) * stride;
50
51     // Prefix sum over the last element of each independent block
52     #pragma omp task weakwait \
53     depend(weakinout: data[substart:N-substart])
54     prefix_sum(data + substart, N - substart, TS, TS*stride);
55
56     // Accumulate the value of the last element of each independent block
57     // over the elements of the following block
58     for (long i = substart; i+stride < N; i += TS*stride) {
59         long size = min(TS*stride, N-i);
60
61         #pragma omp task depend(in: data[i]) \
62         depend(inout: data[i+stride:size-stride])
63         for (long j = stride; j < size; j += stride)
64             data[i+j] += data[i];
65     }
66 }

```

Listing 7. A quicksort followed by a prefix sum, both implemented recursively with weakwaits and weak dependencies.

the final value of those positions. Next, we instantiate one task per block that accumulates over its elements the contribution of the last element of its previous block. Notice that, except for the base case and the last operation, the code does not access the data. For this reason, their corresponding tasks are the only ones that do not use weak dependencies.

The weakwait clause that we use in the quicksort implementation allows the quicksort to release dependencies at the granularity of the base case. The prefix sum, since it uses weak dependencies for all but its leaf tasks, is able to propagate fine grained dependencies outwards. When we combine both algorithms, the innermost tasks of each algorithm end up connecting through fine-grained dependencies.

Figure 7 shows a small section of the execution timeline of the benchmark. The bottom part corresponds to the code already shown, and the top to the code with regular taskwaits and regular dependencies. The horizontal axis corresponds to time, and each colored horizontal bar to a thread. The colors indicate the kind of task that a given thread is executing at a given time.

Notice on the bottom half that the fine grained control of dependencies allows tasks from both algorithms to be executed concurrently. On one hand, the weakwait of the quicksort propagates the fine-grained dependencies to its successors. On the other hand, the weak dependencies allow the prefix sum to instantiate its subtasks and to link them to their predecessors.

When we disable either the weakwait of the quicksort tasks, or the weak dependencies of the prefix sum, this is no longer possible. Instead, the prefix sum must wait for the full quicksort to finish.

## IX. CONCLUSIONS

The OpenMP tasking model has support for task nesting and dependencies. Programming algorithms that have dependencies using a top-bottom approach naturally leads to task nesting. However, the current mechanisms to coordinate the correct handling of dependencies across nesting levels restrict the exploitation of parallelism. This is a consequence of each task defining an isolated domain of dependencies for its direct subtasks.

In this paper we have made proposals to enhance the tasking model of OpenMP in a way that breaks the isolation between the dependency domains. First, we have proposed an alternative to the *taskwait* directive that has different semantics. While it does not directly solve part of the initial problem, it serves as basis on which to build the following proposal. In addition, this contribution has uses outside of the scope of improving the integration between dependencies and nesting.

Our second proposal solves the propagation of dependencies from inner tasks to outer tasks. This new capability enables the release of dependencies at a finer-grain than it was previously possible. Hence, it has the potential to uncover more parallelism.

Our third contribution solves the propagation of dependencies from outer tasks to inner tasks. Our solution consists of two improvements. First, we allow tasks to start before the

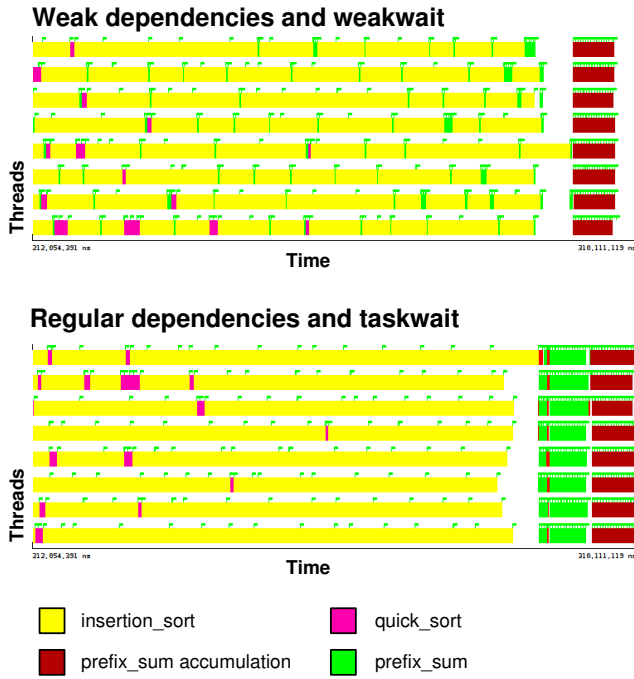


Fig. 7. Detail of the execution timeline of a quicksort followed by a prefix sum (top) with weakwait and weak dependencies, and (bottom) with taskwait and regular dependencies.

dependencies that are only needed for their subtasks have been satisfied. This way, the subtasks can be instantiated earlier. And second, we allow to propagate the fine-grained release of dependencies to subtasks.

By combining these two last contributions, dependencies can cross the boundaries initially set up by the nesting contexts. The resulting behavior is equivalent to performing all the dependency analysis on a single domain of dependencies. Previously, to achieve a similar effect in the general case required avoiding nesting. However, that approach reduces the programmability and also restricts the instantiation of tasks to a single generator. Our proposal allows extracting the same amount of parallelism, without loss of programmability and without the loss of the parallel generation of work that is achievable through nesting.

We have demonstrated that the techniques help in creating work in parallel and allow to uncover more and more distant parallelism. This information is also useful to produce better schedulings. As the number of cores in processors increases, we expect that the work per core will decrease. Our preliminary experiments have shown that our improvements are helpful in such scenarios.

## X. FUTURE WORK

When a programmer uses a top-down approach to annotate an application, the top-level tasks are first annotated and then each top-level task can be recursively split into subtasks. In this way, after a few coarse grained top-level tasks have been instantiated, the runtime has access to a general overview of how data will be used in the distant future. We plan to use

this information to improve scheduling decisions that maximize data reuse but minimize the size of the task scheduling window.

We also plan to explore how the techniques proposed in this paper can be extended to improve task reductions [13] in the presence of task nesting. Moreover, we will also study the potential of the weak in/out clause on the OmpSs@cluster [14] distributed programming environments. Currently, the dataset of a distributed task is limited by the physical memory of a node. Using weak dependencies we plan to overcome this limitation by replacing the eager copy of the whole dataset by a lazy copy of the subset required by each subtask.

## ACKNOWLEDGMENTS

This work is supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology (project TIN2015-65316-P) and by the Generalitat de Catalunya (grant 2014-SGR-1051).

## REFERENCES

- [1] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.
- [2] C. E. Leiserson, "The Cilk++ concurrency platform," *The Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, 2010.
- [3] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, "Scheduling dense linear algebra operations on multicore processors," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 1, pp. 15–44, Jan. 2010.
- [4] E. Ayguade, N. Copt, A. Duran, J. Hoefflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of OpenMP tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, March 2009.
- [5] OpenMP Architecture Review Board, "OpenMP application program interface version 3.0," May 2008.
- [6] A. Duran, R. Ferrer, E. Ayguadé, R. M. Badia, and J. Labarta, "A proposal to extend the OpenMP tasking model with dependent tasks," *International Journal of Parallel Programming*, vol. 37, no. 3, pp. 292–305, 2009.
- [7] OpenMP Architecture Review Board, "OpenMP application programming interface 4.5," Nov. 2015.
- [8] J. M. Perez, R. M. Badia, and J. Labarta, "Handling task dependencies under strided and aliased references," in *Proc. of the 24th ACM Int. Conf. on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 263–274.
- [9] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, pp. 173–193, 2011-03-01 2011.
- [10] N. Rajovic, A. Rico, F. Mantovani, D. Ruiz, J. Oriol Vilarrubi, C. Gomez, L. Backes, D. Nieto, H. Servat, X. Martorell, J. Labarta, E. Ayguade, C. Adeniyi-Jones, S. Derradji, H. Gloaguen, P. Lanucara, N. Sanna, J.-F. Méhaut, K. Pouget, B. Videau, E. Boyer, M. Allalen, A. Auweter, D. Tafani, D. Brayford, D. Brömmel, R. Halver, J. H. Meinke, R. Beivide, M. Benito, E. Vallejo, M. Valero, and A. Ramirez, "The Mont-Blanc Prototype: An Alternative Approach for HPC Systems," in *Proceedings of the 2016 IEEE/ACM conference on Supercomputing, SC'16*, 2016.
- [11] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, no. 7, pp. 321–, Jul. 1961.
- [12] G. E. Blelloch, "Prefix sums and their applications," *Synthesis of Parallel Algorithms*, Tech. Rep., 1990.
- [13] J. Ciesko, S. Mateo, X. Teruel, X. Martorell, E. Ayguadé, J. Labarta, A. Duran, B. R. de Supinski, S. Olivier, K. Li, and A. E. Eichenberger, *Towards Task-Parallel Reductions in OpenMP*. Cham: Springer International Publishing, 2015, pp. 189–201.
- [14] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with omps," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 555–566.